International Journal of Research
Available at http://internationaljournalofresearch.org/
p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 02 Issue 09
October 2015

Implementation of Automatic Test Packet Generation

N.Venkatadri #1, A.Chaitanya #2
#1 Assoc. Professor, Dept. Of CSE, SKR College Of Engineering And Technology, Kondurusatram, Manubolu,Nellore,AP
#2 PG Student, Dept. Of CSE, SKR College Of Engineering And Technology, Kondurusatram,Manubolu,Nellore,AP.

Abstract:

Networks are getting larger and more complex, yet administrators rely on rudimentary tools such as and to debug problems. We propose an automated and systematic approach for testing and debugging networks called "**Automatic Test Packet Generation**" (ATPG). ATPG reads router configurations and generates a device-independent model. The model is used to generate a minimum set of test packets to (minimally) exercise every link in the network or (maximally) exercise every rule in the network. Test packets are sent periodically, and detected failures trigger a separate mechanism to localize the fault. ATPG can detect both functional (e.g., incorrect firewall rule) and performance problems (e.g., congested queue). ATPG complements but goes beyond earlier work in static checking (which cannot detect liveness or performance faults) or fault localization (which only localize faults given liveness results). We describe our prototype ATPG implementation and results on two real-world data sets: Stanford University's backbone network and Internet2. We find that a small number of test packets suffices to test all rules in these networks. For example, 4000 packets can cover all rules in Stanford backbone network, while 54 are enough to cover all links. Sending 4000 test packets 10 times

per second consumes less than 1% of link capacity. ATPG code and the data sets are publicly available.

Index Terms—Data plane analysis, network troubleshooting, test packet generation.

1.INTRODUCTION

It is notoriously hard to debug networks. Every day, network engineers wrestle with router misconfigurations, fiber cuts, faulty interfaces, mislabeled cables, software bugs, intermittent links, and a myriad other reasons that cause networks to misbehave or fail completely. Network engineers hunt down bugs using the most rudimentary tools (e.g., SNMP) and track down root causes using a combination of accrued wisdom and intuition. Debugging networks is only becoming harder as networks are getting *bigger* (modern data centers may contain 10 000 switches, a campus network may serve 50 000 users, a 100-Gb/s long-haul

International Journal of Research
Available at http://internationaljournalofresearch.org/
p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 02 Issue 09
October 2015

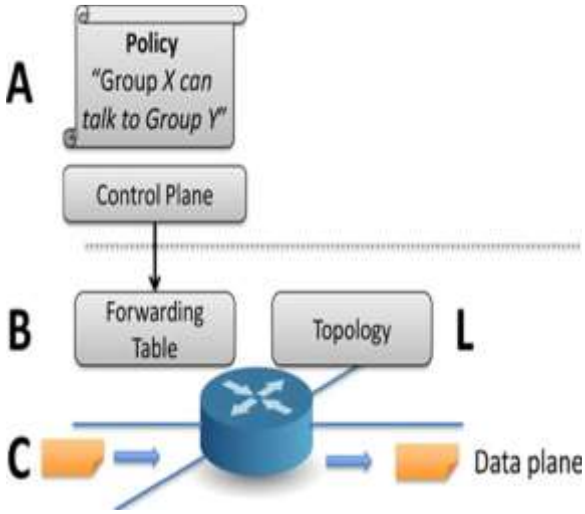


Fig.1. Static versus dynamic checking: A policy is compiled to forwarding state,which is then executed by the forwarding plane. Static checking (e.g., [16]) confirms that . Dynamic checking (e.g., ATPG in this paper) confirms that the topology is meeting liveness properties *and* that .link may carry 100000 flows) and are getting *more complicated* (with over 6000 RFCs, router software is based on millions of lines of source code, and network chips often contain billions of gates). It is a mall wonder that network engineers have been labeled "masters of complexity" [32]. Consider two examples. *Example 1:* Suppose a router with a faulty line card starts dropping packets silently. Alice, who administers 100 routers, receives a ticket from several unhappy users complaining about connectivity. First, Alice examines each router to see if the configuration was changed recently and concludes that the configuration was untouched. Next, Alice uses her knowledge of the topology to triangulate the faulty device with and . Finally, she calls a colleague to replace the line card.

We can think of the controller compiling the policy (A) into device-specific *configuration* files (B), which in turn determine the forwarding behavior of each packet (C). To ensure the net-work behaves as designed, all three steps should remain consis-tent at all times, i.e., $A–B–C$. In addition, the topology, shown to the bottom right in the figure, should also satisfy a set of livenessproperties L. Minimally, L requires that sufficient links and nodes are working; if the control plane specifies that a laptop can access a server, the desired outcome can fail if links fail. Lcan also specify performance guarantees that detect flaky links.

Recently, researchers have proposed tools to check that $A–B$, enforcing consistency between *policy* and the *config-uration*[7], [16], [25], [31]. While these approaches canfind(or prevent) software logic errors in the control plane, they are *not* designed to identify liveness failures caused by failed linksand routers, bugs caused by faulty router hardware or software, or performance problems caused by network congestion. Such failures require checking for L and whether $B–C$. Alice's first problem was with L (link not working), and her second problem was with $B–C$ (low level token bucket state not reflecting policy for video bandwidth).

In fact, we learned from a survey of 61 network operators (see Table I in Section II) that the two most common causes of net-work failure are hardware failures and software bugs, and that problems manifest themselves *both* as reachability failures and throughput/latency degradation. Our goal is to automatically detect these types of failures.

The main contribution of this paper is what we call an Automatic Test Packet Generation (ATPG) framework that *automatically* generates a minimal set of packets to test the liveness ofthe underlying topology *and* the congruence between data plane state and configuration specifications. The tool can also automatically generate packets to test *performance* assertions such as packet latency. In Example 1, instead of Alice manually deciding which `Ping` packets to send, the tool does so periodically on her behalf. In Example 2, the tool determines that it must send packets with certain headers to "exercise" the video queue, and then determines that these packets are being dropped.

ATPG detects and diagnoses errors by independently and exhaustively *testing* all forwarding entries, firewall rules, and any packet processing rules in the network. In ATPG, test packets are generated algorithmically from the device configuration files and FIBs, with the minimum number of packets required for complete coverage. Test packets are fed into the network so that every rule is exercised directly from the data plane. Since ATPG treats links just like normal forwarding rules, its full coverage guarantees testing of every link in the network. It can also be specialized to generate a minimal set of packets that merely test every link for network liveness. At least in this basic form, we feel that ATPG or some similar technique is fundamental to networks: Instead of *reacting* to failures, many network operators such as Internet2 [14] *proactively* check the health of their network using pings between all pairs of sources. How-ever, all-pairs `Ping` does not guarantee testing of all links andhas b een found to be unscalable for large networks such as PlanetLab [30].

Organizations can customize ATPG to meet their needs; for example, they can choose to merely check for network liveness (link cover) or check every rule (rule cover) to ensure security policy. ATPG can be customized to check only for reachability or for performance as well. ATPG can adapt to constraints such as requiring test packets from only a few places in the network or using special routers to generate test packets from every port. ATPG can also be tuned to allocate more test packets to exercise more critical rules. For example, a healthcare network may dedicate more test packets to Firewall rules to ensure HIPPA compliance.

We tested our method on two real-world data sets the back-bone networks of Stanford University, Stanford, CA, USA, and Internet2, representing an enterprise network

and a nationwide ISP. The results are encouraging: Thanks to the structure of real world rule sets, the number of test packets needed is surprisingly small. For the Stanford network with over 757 000 rules and more than 100 VLANs, we only need 4000 packets to exercise all forwarding rules and ACLs. On Internet2, 35 000 packets suffice to exercise all IPv4 forwarding rules. Put another way, we can check every rule in every router on the Stanford backbone 10 times every second by sending test packets that consume less than 1% of network bandwidth. The link cover for Stanford is even smaller, around 50 packets, which allows proactive liveness testing every millisecond using 1% of net-work bandwidth.

The contributions of this paper are as follows:

1) a survey of network operators revealing common failures and root causes (Section II);

2) a test packet generation algorithm (Section IV-A);

3) a fault localization algorithm to isolate faulty devices and rules (Section IV-B);
4) ATPG use cases for functional and performance testing (Section V);

5) evaluation of a prototype ATPG system using rulesets collected from the Stanford and Internet2 backbones

II. CURRENT PRACTICE

To understand the problems network engineers encounter, and how they currently troubleshoot them, we invited sub-scribers to the NANOG[1] mailing list to complete a survey in May–June 2012. Of the 61 who responded, 12 administer small networks (<1 k hosts), 23 medium networks (1 k–10 k hosts), 11 large networks (10 k–100 k hosts), and 12 very large networks (>100 k hosts). All responses (anonymized) are reported in [33] and are summarized in Table I. The most relevant findings are as follows.

Symptoms: Of the six most common symptoms, four cannotbe detected by static checks of the type A–B (throughput/ latency, intermittent connectivity, router CPU utilization, con-gestion) and require ATPG-like dynamic testing. Even the re-maining two failures (reachability failure and security Policy Violation) may require dynamic testing to detect forwarding plane failures.

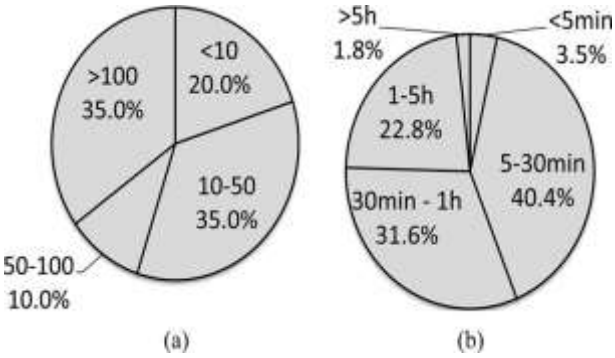


Fig. 1. Reported number of (a) network-related tickets generated per month and (b) time to resolve a ticket.

Cost of troubleshooting:
Two metrics capture the cost ofnetwork debugging—the number of network-related tickets per month and the average time consumed to resolve a ticket (Fig. 1). There are 35% of networks that generate more than 100 tickets per month. Of the respondents, 40.4% estimate it takes under 30 min to resolve a ticket. However, 24.6% report that it takes over an hour on average.

Tools: Table II shows that ping, traceroute, and SNMP are by far the most popular tools. When asked what the ideal tool for network debugging would be, 70.7% reported a desire for automatic test generation to check performance and correctness. Some added a desire for "long running tests to detect jitter or intermittent issues," "real-time link capacity monitoring," and "monitoring tools for network state."

In summary, while our survey is small, it supports the hypoth-esis that network administrators face complicated symptoms and causes. The cost of debugging is nontrivial due to the frequency of problems and the time to solve these problems. Classical tools such as ping and traceroute are still heavily used, but administrators desire more sophisticated tools.

III. NETWORK MODEL

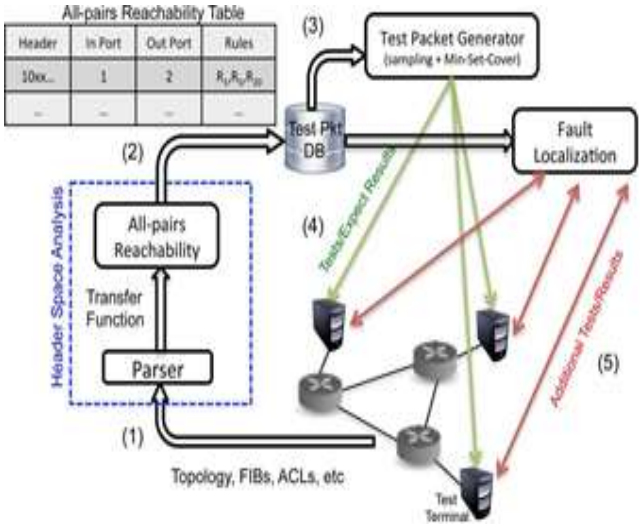
ATPG uses the *header space* framework—a geometric model of how packets are processed we described in [16] (and used in [31]). In header space, protocol-specifi c meanings associ-ated with headers are ignored: A header is viewed as a flat se-quence of ones and zeros. A header is a point (and a flow is a region) in the $\{0,1\}^L$ space, where L is an upper bound on header length. By using the header space framework, we ob-tain a unified, vendor-independent, and protocol-agnostic model of the network[2] that simplifies the packet generation process significantly.

```
function NETWORK(packets, switches, Γ)
  for pk0 ∈ packets do
    T ←FIND_SWITCH(pk0.p, switches)
    for pk1 ∈ T(pk0) do
      if pk1.p ∈ EdgePorts then
        #Reached edge
        RECORD(pk1)
      else
        #Find next hop
        NETWORK(Γ(pk1), switches, Γ)
```

Fig:2. Life of a packet: repeating $\mathcal{T}$ and τ until the packet reaches its destination or is dropped.

International Journal of Research
Available at http://internationaljournalofresearch.org/
p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 02 Issue 09
October 2015

IV. ATPG SYSTEM



Based on the network model, ATPG generates the minimal number of test packets so that every forwarding rule in the net-work is exercised and covered by at least one test packet. When an error is detected, ATPG uses a fault localization algorithm to determine the failing rules or links.

Fig. 3 is a block diagram of the ATPG system The system first collects all the forwarding state from the network (step 1). This usually involves reading the FIBs, ACLs, and config files, as well as obtaining the topology. ATPG uses Header Space Analysis [16] to compute reachability between all the test terminals (step 2).

Fig.3. ATPG system block diagram.

A. Test Packet Generation

1) Algorithm: We assume a set of *test terminals* in the net-work can send and receive test packets. Our goal is to generate a set of test packets to exercise *every* rule in *every* switch function, so that *any* fault will be observed by at least one test packet. This is analogous to software test suites that try to test every possible branch in a program. The broader goal can be limited to testing every link or every queue.

When generating test packets, ATPG must respect two key constraints: 1) *Port:* ATPG must only use test terminals that are available; 2) *Header:* ATPG must only use headers that each test terminal is permitted to send. For example, the network admin-iterator may only allow using a specific set of VLANs. Formally, we have the following problem.

Problem 1 (Test Packet Selection): For a network with theswitch functions, $T_1..T_n$, and topology function, Γ, determini the minimum set of test packets to exercise all reach-able rules, subject to the port and header constraints.

ATPG chooses test packets using an algorithm we call *TestPacket Selection* (TPS). TPSfirstfinds all *equivalent classes* between each pair of available ports. An equivalent class is a set of packets that exercises the same combination of rules.

V. RELATED WORK

We are unaware of earlier techniques that automatically generate test packets from configurations. The closest related works we know of are offline tools that check invariants in net-works. In the control plane, NICE [7] attempts to exhaustively cover the code paths symbolically in controller applications with the help of simplified switch/host models. In the data plane, Anteater [25] models invariants as booleansatisfiability problems and checks them against configurations with a SAT solver. Header Space Analysis [16] uses a geometric model to check reachability, detect loops, and verify slicing. Recently, SOFT [18] was proposed to verify consistency between different Open Flow agent implementations that are responsible for bridging control and data planes in the SDN context. ATPG complements these checkers by directly *testing* the data plane and covering a significant set of dynamic or performance errors that cannot otherwise be captured.

End-to-end probes have long been used in network fault diagnosis in work such as [8]–[10], [17], [23], [24], [26]. Recently, mining low-quality, unstructured data, such as router confi durations and network tickets, has attracted interest [12], [21], [34]. By contrast, the primary contribution of ATPG is not fault localization, but determining a compact set of end-to-end measurements that can cover every rule or every link. The mapping between Min-Set -Cover and network monitoring has been previously explored in [3] and [5]. ATPG improves the detection granuality to the rule level by employing router configuration and data plane information. Furthermore, ATPG is not limited to liveness testing, but can be applied to checking higher level properties such as performance.

There are many proposals to develop a measurement-friendly architecture for networks [11], [22], [28], [35]. Our approach is complementary to these proposals: By incorporating input and port constraints, ATPG can generate test packets and injection points using existing deployment of measurement devices.

Our work is closely related to work in programming lan-guages and symbolic debugging. We made a preliminary at-tempt to use KLEE [6] and found it to be 10 times slower than even the unoptimized header space framework. We speculate that this is fundamentally because in our framework we directly *simulate* the forward path of a packet instead of *solving constraints* using an SMT solver. However, more work is requiredto understand the differences and potential opportunities.

VI. CONCLUSION

Testing liveness of a network is a fundamental problem for ISPs and large data center operators. Sending probes between every pair of edge ports is neither exhaustive nor scalable [30]. It suffices to find a

minimal set of end-to-end packets that traverse each link. However, doing this requires a way of abstracting across device specific configuration files (e.g., header space), generating headers and the links they reach (e.g., all -pairs reach-ability), and finally determining a minimum set of test packets (Min- Set-Cover). Even the fundamental problem of automat-ically generating test packets for efficient liveness testing re-quires techniques akin to ATPG.

ATPG, however, goes much further than liveness testing with the same framework. ATPG can test for reachability policy (by testing all rules including drop rules) and performance health (by associating performance measures such as latency and loss with test packets). Our implementation also augments testing with a simple fault localization scheme also constructed using the header space framework. As in software testing, the formal model helps maximize test coverage while minimizing test packets. Our results show that all forwarding rules in Stanford backbone or Internet2 can be exercised by a surprisingly small number of test packets (<4000 for Stanford, and <40000 for Internet2).

Network managers today use primitive tools such as ping

and traceroute . Our survey results indicate that they are eager for more sophisticated tools. Other fields of engineering indi-cate that these desires are not unreasonable: For example, both the ASIC and software design industries are buttressed by bil-lion-dollar tool businesses that supply techniques for both static (e.g., design rule) and dynamic (e.g., timing) verification. In fact, many months after we built and named our system, we dis-covered to our surprise that ATPG was a well-known acronym in hardware chip testing, where it stands for Automatic Test *Pat-tern* Generation [2]. We hope network ATPG will be equallyuseful for automated dynamic testing of production networks.

REFERENCES

[1] "ATPG code repository," [Online]. Available: http://eastzone.github.com/atpg/

[2] "Automatic Test Pattern Generation," 2013 [Online]. Available: http://en.wikipedia.org/wiki/Automatic_test_pattern_generation

[3] P. Barford, N. Duffield, A. Ron, and J. Sommers, "Network perfor-mance anomaly detection and localization," in *Proc. IEEE INFOCOM*, Apr. , pp. 1377–1385.

[4] "Beacon," [Online]. Available: http://www.beaconcontroller.net/

[5] Y. Bejerano and R. Rastogi, "Robust monitoring of link delays and faults in IP networks," *IEEE/ACM Trans. Netw.*, vol. 14, no. 5, pp. 1092–1103, Oct. 2006.

[6] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. OSDI*, Berkeley, CA, USA, 2008, pp. 209–224.

[7] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford, "A NICE way to test OpenFlow applications," in *Proc. NSDI*, 2012, pp. 10–10.

[8] A. Dhamdhere, R. Teixeira, C. Dovrolis, and C. Diot, "Netdiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data," in *Proc. ACM CoNEXT*, 2007, pp. 18:1–18:12..

[9] N. Duffield, "Network tomography of binary network performance characteristics," *IEEE Trans. Inf. Theory*, vol. 52, no. 12, pp. 5373–5388, Dec. 2006.

[10] N. Duffield, F. L. Presti, V. Paxson, and D. Towsley, "Inferring link loss using striped unicast probes," in *Proc. IEEE INFOCOM*, 2001, vol. 2, pp. 915–923. 65

[11] N. G. Duffield and M. Grossglauser, "Trajectory sampling for direct traffic observation," *IEEE/ACM Trans. Netw.*, vol. 9, no. 3, pp. 280–292, Jun. 2001.

[12] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," in *Proc. ACMSIGCOMM*, 2011, pp. 350–361.

[13] "Hassel, the Header Space Library," [Online]. Available: https://bit-bucket.org/peymank/hassel-public/

[14] Internet2, Ann Arbor, MI, USA, "The Internet2 observatory data col-lections," [Online]. Available: http://www.internet2.edu/observatory/ archive/data-collections.html

[15] M. Jain and C. Dovrolis, "End-to-end available bandwidth: Measure-ment methodology, dynamics, and relation with TCP throughput," *IEEE/ACM Trans. Netw.*, vol. 11, no. 4, pp. 537–549, Aug. 2003.

[16] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. NSDI*, 2012, pp. 9–9.

[17] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren, "IP fault localization via risk modeling," in *Proc. NSDI*, Berkeley, CA, USA, 2005, vol. 2, pp. 57–70.

[18] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic, "A SOFT way for OpenFlow switch interoperability testing," in *Proc.ACM CoNEXT*, 2012, pp. 265–276.

[19] K. Lai and M. Baker, "Nettimer: A tool for measuring bottleneck link, bandwidth," in *Proc. USITS*, Berkeley, CA, USA, 2001, vol. 3, pp. 11–11.

[20] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proc. Hotnets*, 2010, pp. 19:1–19:6.

[21] F. Le, S. Lee, T. Wong, H. S. Kim, and D. Newcomb, "Detecting network-wide and router-specific misconfigurations through data mining," *IEEE/ACM Trans. Netw.*, vol. 17, no. 1, pp. 66–79, Feb. 2009.

[22] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani, "iplane: An information plane for distributed services," in *Proc. OSDI*, Berkeley, CA, USA, 2006, pp. 367–380.

[23] A. Mahimkar, Z. Ge, J. Wang, J. Yates, Y. Zhang, J. Emmons, B. Huntley, and M. Stockert, "Rapid detection of maintenance induced changes in service performance," in *Proc. ACM CoNEXT*, 2011, pp. 13:1–13:12.

[24] A. Mahimkar, J. Yates, Y. Zhang, A. Shaikh, J. Wang, Z. Ge, and C. T. Ee, "Troubleshooting chronic conditions in large IP networks," in *Proc. ACM CoNEXT*, 2008, pp. 2:1–2:12.

[25] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with Anteater," *Comput. Commun.Rev.*, vol. 41, no. 4, pp. 290–301, Aug. 2011.

[26] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C.-N. Chuah, Y. Ganjali, and C. Diot, "Characterization of failures in an operational ip backbone network," *IEEE/ACM Trans. Netw.*, vol. 16, no. 4, pp. 749–762, Aug. 2008.

[27] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *Comput. Commun. Rev.*, vol. 38, pp. 69–74, Mar. 2008.

[28] "OnTimeMeasure," [Online]. Available: http://ontime.oar.net/

[29] "Open vSwitch," [Online]. Available: http://openvswitch.org/

[30] H. Weatherspoon, "All-pairs ping service for PlanetLab ceased," 2005 [Online]. Available: http://lists.planet-lab.org/pipermail/users/2005-July/001518.html

[31] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proc. ACM SIGCOMM*, 2012, pp. 323–334.

[32] S. Shenker, "The future of networking, and the past of protocols," 2011 [Online]. Available: http://opennetsummit.org/archives/oct11/shenker-tue.pdf

[33] "Troubleshooting the network survey," 2012 [Online]. Available: http://eastzone.github.com/atpg/docs/NetDebugSurvey.pdf

[34] D. Turner, K. Levchenko, A. C.Snoeren, and S. Savage, "California fault lines: Understanding the causes and impact of network failures," *Comput. Commun. Rev.*, vol. 41, no. 4, pp. 315–326, Aug. 2010.

[35] P. Yalagandula, P. Sharma, S. Banerjee, S. Basu, and S.-J. Lee, "S3: A scalable sensing service for monitoring large networked systems," in *Proc. INM*, 2006, pp. 71–76.

Guide Profile:

Mr. N.Venkatadri was born in Andhra Pradesh, India. He is working as Asso.Prof., M.Tech Department of CSE, SKR College Of Engineering and Technology, Konduru Satram, Manubolu, Nellore(DT).

Student Profile:

A.Chaitanya was born in Andhra Pradesh, India. He received B.Tech Degree from JNTU Ananthapur, Nellore (DT). I am pursuing M.Tech Degree in CSE from JNTU Ananthapur.